UNIVERSITY OF GLASGOW

COMPUTING DEPARTMENT

MULTUM PASCAL COMPILER MEMO NO. 1 (20/12/72)

A RUN TIME ENVIRONMENT FOR MULTUM PASCAL

1 STACK FRAME STRUCTURE

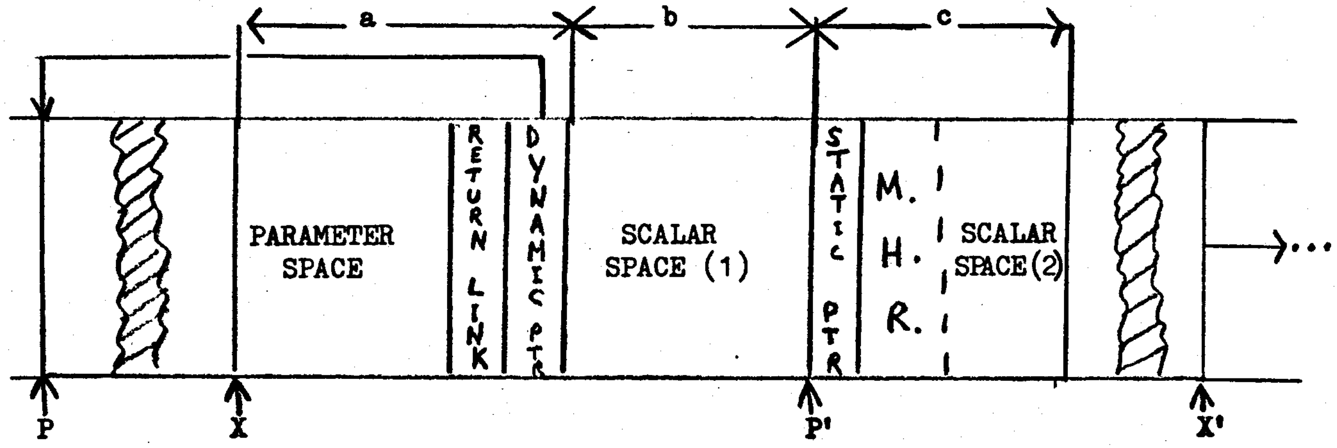


FIGURE 1

Figure 1 gives the structure of the stack frame of a procedure activation. The procedure requires (a + b + c) words for its parameter, link and scalar spaces, where $(a + b + c) < 257$ due to hardware limitations.

The parameter space occupies $(a - 2) \geq 0$ words, and these are set up before entry to the procedure to contain either the values of <u>const</u> parameters or the addresses of <u>var</u> parameters.

The two words which follow contain respectively the return address link to the calling routine and the dynamic chain pointer (the value of the P register in the calling routine).

The following (b + c) words encompass the rest of the link space, and the whole of the scalar space. Choose $b > 0$ only if it would otherwise be necessary to have $C > 127$. Suppose g memory-held registers are necessary to address all of global storage. Then (g + 1) further words of link space are reserved, in the first (g + 1) memory-held registers. M0, the word P points to, is used to hold the static chain pointer (the value of the P register in the most recent activation of the textually enclosing procedure) and is used to access non-local variables. M1 through Mg are used to access global variables. The remaining (b + c - g - 1) words contain the scalar space of the procedure - that is, the local scalar variables and pointer words to the local structured variables. M6 through M15 hold the "partial results stack".

Storage above the scalar space is allocated to structures (a fixed amount for a given procedure). The X register is dedicated to pointing to the next free word (the "top of stack").

2 ACCESS TO DATA

(a) local variables and <u>const</u> parameters are directly-addressed using mode Pp, where p is the position of the datum relative to M0,

$$-128 \leq p \leq 127 \;;$$

(b) var parameters are indirectly addressed using mode P_pI ;

(c) global variables are directly addressed using mode MmD, where m and D are chosen so that $0 \leq D \leq 63$ and

$$(Mm) + D = \text{required address},$$

the memory held registers having been set up on entry to the procedure so that

$$M1 = \text{base address of global area}$$
$$M(i+1) = M(i) + 64, \quad i = 1,\ldots,(g-1);$$

(d) non-local quantities other than globals are addressed directly, using the same modes as locals, P being set to point to the correct stack frame by following the static chain pointer (Y is used to preserve the local value of P during this operation):

```
LDRY  P ; {LODP  POI }*;  <access in P  mode> ;  EXRY  P .
```

3 PROCEDURE ACTIVATION SEQUENCES

A procedure is activated and deactivated by three code sequences at the point of call, at the entry to the body of the procedure, and at the exit from the body of the procedure.

(a) call:

```
<code to stack parameter values or addresses>
<code to set up the static link>
      LINK  ZX +
      JUMP  S O I
  <address of the procedure>
```

Here, <code to set up the static link> is one of three sequences:

(i) if the called procedure is local to the calling procedure, use

```
LDRB  P ;
```

(ii) if the called procedure is declared at the same textual level as the calling procedure, use

```
SETB   P O ;
```

(iii) if the called procedure is declared in an outer textual level, n levels out say, use

```
SETB   P O }
SETB   Z B }   repeated  n  times
```

(b) entry:

```
LDRA   P
STAS   Z  X  +  / SAVE DYNAMIC PTR
SETA   L  b }  omit  if  b = 0
ADRX   A    }  ADMX  L  b  if  b < 16
LODP   Z  X
STBS   P  O  / SAVE STATIC PTR
SETA   L  c }  omit if  c = 0
ADRX   A    }  ADMX  Lc  if  c < 16
<code to set up global pointers>
```

Here, <code to set up global pointers> is one of two alternatives:

(i) no code is needed if there is no global space or if no globals are accessed;

(ii) if g registers are required to span the global space, the sequence needed is

```
SETA  MO 1
STAS  P1
ADDA  L 64 }  repeated for  i = 2,..., g
STAS  Pi   }  if  g > 1 .
```

(c) exit:

```
LDRA  P
SUBA  L  (a + b)
LDRX  A
LDRP  A
SETB  P  (a - 2) ;  ADMB  L 1
LODP  P  (a - 1) I
EXRB  S
```

4 BOOLEAN EXPRESSIONS

(a) conjunctions:

```
compile   x ∧ y ∧ ... ∧ z  as
          <deal with  x>
          SFCA                    {skip if false}
          JUMP  S 2
          JUMP  S 0 I
          <address of false condition code>
          <deal with y>
          SFCA
          JUMP   S  2
          JUMP   S 0 I
          <address of false condition code>
               ⋮
          <deal with z>
          SFCA
          JUMP  S2
          JUMP  S 0 I
          <address of false condition code>
          <true condition code>
          <false condition code>
```

(b) disjunctions:

compile x V y V ... V z in the same manner as the conjunction above, reversing the roles of 'true' and 'false'.

(c) relations:

if a Boolean expression x is of the simple form p Ⓡ q where Ⓡ is one of the arithmetic relations, translate

```
<deal with  x>     into     <evaluate  p>
                            <subtract  q>
```

and note that the relevant condition for the next SFCA or STCA is $\textcircled{R}$0, i.e. where $\textcircled{R}$ is condition code is

$>$	PP
$<$	NN
$\geq$	PZ
$\leq$	NZ
$=$	ZZ
$\neq$	ZZ, but reverse

the test (change SFCA to STCA and vice versa).

(d) If the $<\left\{\begin{matrix}\text{true}\\ \text{false}\end{matrix}\right\}$ condition code> is <u>goto</u> label and the label is in the same scope as the <u>goto</u>, replace all sequences of the form

```
JUMP   S O I
<address of {true/false} condition code>
```

by sequences of the form

```
JUMP   S O I
<address of the label>
```

Note that this is a more sophisticated optimisation, not for version 1 unless very easy to do!

5 CASE STATEMENTS

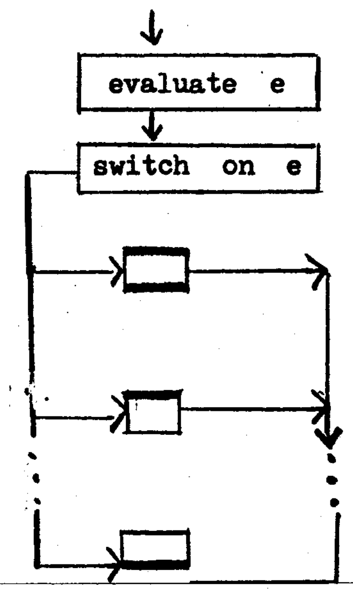
Translate a statement of the form

<u>case</u> e <u>of</u> . . . . <u>end</u>

into code of the form

```
    <evaluate  e>
    LSLA  L1
    SETB  S1I
    JUMP  AB
    <address of jump table t_1>
    <.... coding of component statements ....>
t_i: JUMP  S O I                                repeated for
    <address of component labelled "1">        each value
       :                                        in range of  e
```

Flowchart is



6 FUNCTIONS

The stack frame structure of a function activation is the same as that for a procedure activation, except that the two scalar space words beneath the static pointer are reserved to hold the value of the function.

7 VARIABLE STORAGE STRUCTURES (See figure 2)

(a) Simple variables are stored as one or more consecutive words of scalar space, either in the appropriate procedure block or in the global block.

type	size (words)
integer	1
long integer	2
real	2
char	1
Boolean	1
powerset	4

(b) Array variables consist of two parts. One is the array word, stored in the scalar space and holding the virtual address of the array base (the [0,0, ..., 0] th element). The second part is the array proper, stored in row major order in the structure space. Array elements are the same size as the corresponding simple variables.

(c) Record variables consist of two parts, like arrays. The record word is stored in the scalar space and holds the virtual address of the first word of the record area proper, positioned in the structure space.